CAPSL Intermediate Language*

G. Denker and J. Millen
Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
{denker,millen}@csl.sri.com

1 Introduction

The acronym "CAPSL" stands for "Common Authentication Protocol Specification Language." It is a high-level language for cryptographic authentication and key distribution protocols that could be used as the input format for a variety of formal analysis techniques. The core of a CAPSL specification is a message list like the ones that are often used to present cryptographic protocols in articles and textbooks.

Background

Security analysis tools, whether based on model checking, belief logic, or inductive verification techniques, typically represent protocols in a tool-specific language. Some tool developers have seen the need for translators or compilers that would accept message-list specifications and produce all or part of the lower-level representation needed by their tools. This was done with an earlier version of CAPSL for the Interrogator tool developed at MITRE [Mil97]; ISL, from which CAPSL initially borrowed much of its style, supporting an application of HOL to an extension of the GNY logic [Bra97]; Casper, [Low98], for the application of FDR using a CSP model-checking approach; and a "Standard Notation" translated to per-process CKT5 specifications [Car94].

The idea of having a single common protocol specification language that could be used as the input format for any formal analysis technique was first presented at the 1996 Isaac Newton Institute Programme on Computer Security, Cryptology, and Coding Theory at Cambridge University. A version of CAPSL very close to the current one was subsequently implemented as an interface to the NRL Protocol Analyzer [BMM99].

CAPSL Intermediate Language

Support for multiple analysis tools is accomplished through the CAPSL Intermediate Language (CIL). The idea is illustrated in Figure 1, which mentions a few tools – not intended to be a complete list – by way of example. CAPSL is parsed and translated to CIL, and there are different translators from CIL to whatever form is required for each tool. The translator from CAPSL to CIL can deal with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the transitions of the communicating processes.

Translation of CIL to the input language of other tools is viewed as a task for researchers engaged in applying those tools to protocol analysis. We have efforts underway to furnish the tool-specific sub-translators for application of PVS and Maude. PVS is the SRI verification environment, which we are using to support inductive protocol proofs. Our use of Maude is explained below.

The challenge for the design of CIL is to make it general enough and expressive enough to represent a wide range of protocols, and yet at a low enough level to be close to the representation used by most verification or model-checking tools. Many such tools share a specification style

*Supported by DARPA through Air Force Research Laboratory under Contract F30602-98-C-0258.

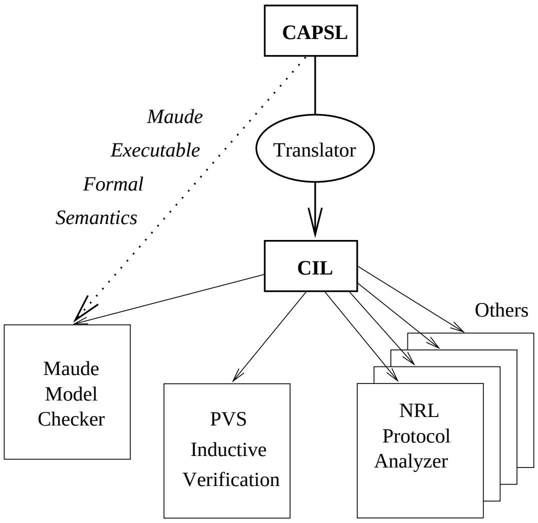

Figure 1: CAPSL Translation Plan

that incorporates state-transition rules specified in a pattern-matching style, with symbolic terms to represent encryption and other computations. There is usually a separate and fairly standard Dolev-Yao-like attacker model.

The meta-notation of Cervesato, et al [Cer99] was adapted for CIL because it appeared to be the simplest and most general formalism that exhibits these features. Their notation, according to the authors, could be regarded as either an extension of multiset rewriting with a kind of existential quantification, or a Horn fragment of linear logic. The logical style of this formalism made it suitable to serve as the language in which to express the semantics of CAPSL. Furthermore, the term-rewriting aspect corresponded well with the analysis approach taken by Denker, Meseguer, and Talcott with Maude [DMT98]. CIL may be regarded as a notational variant of the linear logic formalism in which certain specific conventions have been used to set up protocol models derived from CAPSL specifications.

Semantics in Maude

While CIL has its own rewriting-logic semantics, the semantics of CAPSL lies in its translation to CIL. In this paper we summarize part of the message-list translation in a general state-machine fashion to begin with, and then indicate how it is implemented in Maude.

There is a prototype CAPSL-to-CIL translator in Maude, a multi-paradigm executable specification language based on rewriting logic [Mes92]. The Maude specification of the translator can be understood as the formal semantics of CAPSL in CIL.

We can exploit the reflective capabilities of Maude to also define in Maude the translation from CIL into an executable Maude specification which can be executed, simulated, or model-checked on the Maude rewrite engine [CELM96]. Applying these two pieces of Maude software to a given CAPSL protocol specification, first, to derive the CIL specification, and then, to get an executable protocol specification, yields an executable, formal semantics of CAPSL.

2 CAPSL Basics

A CAPSL specification is made up of three kinds of subspecifications: *type*, *protocol*, and *environment* specifications, usually in that order. Type specifications define cryptographic operators and other functions axiomatically, and are also used to define different types of principals. There is a prelude defining several commonly used operators, such as the familiar abstract symmetric-key and public-key encryption. Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools.

A protocol specification includes a message list. The message list is preceded by programming-language-like type declarations for protocol variables and assumptions about initial conditions. It is followed by a list of security goals.

Here is a small example of a protocol specification, illustrating several of the language features.

```
PROTOCOL One_Message;
IMPORTS USER;
VARIABLES
   A, B: User;
   N: Nonce, CRYPTO;
ASSUMPTIONS
   HOLDS A: B;
MESSAGES
   A -> B: {N}SK(A);
GOALS
   SECRET N;
   BELIEVES B: HOLDS A: N;
END
```

In CAPSL, variables may have properties like CRYPTO that have significance in CIL or the attacker model. (Variables of type Nonce are implicitly assumed to have a FRESH property, but are not necessarily unguessable as indicated by CRYPTO.)

In the message, the nonce N is encrypted using A's secret key. The bracket notation for encryption is syntactic sugar reserved for certain standard operators defined in the prelude.

A specification may import another specification. In this example, the analyst-supplied type specification USER (see Section 4) is imported; it contains the User type and the secret-key function `SK`.

Assumptions and goals may be stated with keywords indicating special state-dependent predicates such as HOLDS and BELIEVES and SECRET. Their semantics can be understood in terms of the CIL model of the protocol.

The message list may also contain equations representing equality tests or assignment statements. It could have begun with a statement `T = {N}SK(A)`, in which case the message could have been written `A -> B: T;` assuming T had been declared. The definition for T could also be placed above in a separate DENOTES section. Such definitions can be invoked implicitly wherever needed.

CAPSL borrows the Casper %-notation. A message field $t\%r$ is constructed by the sender as indicated by t and expected by the receiver to be of the form r. Thus, if the message had been written `A -> B: {N}SK(A)%R;` it would mean that B saves the message field as a variable R but does not decrypt it to obtain N.

The current CAPSL language specification is posted on the CAPSL web site, at `http://www.csl.sri.com/~millen/capsl`. The specification includes an abstract syntax for CAPSL and for CIL, and it is accompanied by several examples of CAPSL specifications.

3 Multiset Term Rewriting

The linear logic formalism uses transition rules of the form

$$F_1, ..., F_k \longrightarrow \exists X_1, ..., X_m : G_1, ..., G_n,$$

where each F_i and G_j is a "fact." Facts are atomic formulas of the form $P(t_1, ..., t_r)$ where P is a predicate symbol and the arguments t_i are terms. A term is constructed from typed constants, variables, and function symbols. Free variables are implicitly universally quantified.

The state of a system can be represented by a multiset of facts. A rule is eligible to fire when the facts on the left side of the rule can be matched with facts in the multiset. When a rule fires, the matching facts in the multiset are removed from it and replaced by the facts on the right side of the rule, instantiated according to the substitution required by the pattern match. Removing a fact from the multiset reduces its multiplicity by one, if it was more than one. Facts in the multiset are typically ground terms (no variables) when finite-state search tools are used.

The existential quantifier in linear logic has a special meaning. Quantified variables are instantiated with fresh (unused) constants. This behavior is used to model generation of nonces.

In protocol modeling, facts are used to express the entrance of a process into a state, or the transmission of a message. In CIL, in particular, a state is represented by a fact `state`(*role, num, terms...*) where a role is a constant corresponding to a principal, *num* is a state label, usually an integer, and the term arguments are the "memory" of the agent in that role and state. A message is a fact `msg`(*sender, receiver, fields...*). Another kind of fact can represent attacker knowledge.

Rules with an empty left side are interpreted as initialization or fact-generating rules. For each role in the protocol, an initial state fact is generated with initially held variables. The rule

$$\longrightarrow \texttt{state('A'},0,A,B),\texttt{state('B'},0,B)$$

creates two facts representing the initial state of two new agents or processes of the protocol, one in role 'A' and the other in role 'B'. The role names are constants taken from the text of the protocol specification, and should be distinguished from the variables A and B, which are free variables whose names could be changed without changing the meaning of the rule. Since A and B are variables of type Principal, this rule can initiate sessions between any pairs of principals.

The message `A -> B: {N}SK(A)` would result in at least two transitions, one for the sender A and one for the receiver B. The A transition would be:

$$\texttt{state('A'},0,A,B) \longrightarrow \exists N : \texttt{state('A'},1,A,B,N),\texttt{msg}(A,B,\texttt{ped}(\texttt{SK}(A),N))$$

where `ped` is the public-key encryption/decryption operator.

4 Type Specification in CAPSL

In order to explain how messages are translated into rewrite rules, we must first introduce two relevant features of type specifications: accessibility and invertibility. A function is *accessible* to a principal if it is not private to some other principal, as a key table might be. A function is *invertible* if some of its arguments can be recovered from its value, given perhaps some additional information such as another argument, or a key.

Accessibility

Consider the following example typespec that defines a subtype User of Principal having a public-key pair.

```
TYPESPEC USER;
TYPES User: Principal;
FUNCTIONS
  Pub(User): Pkey;
  SK(User): Pkey, PRIVATE, CRYPTO;
AXIOMS
  keypair(Pub(U), SK(U));
END;
```

The function that delivers the secret key of a user is given the PRIVATE property to indicate that no user (or the attacker) can invoke this function to obtain another user's secret key. The first argument of a private function is the principal who can evaluate it with that first argument. Thus, a message `A -> B: {X}SK(B);` would be impossible for A to construct *unless* A had previously received the whole term `{X}SK(B)` in a prior message.

Invertibility

A term like $\{X\}K$ is invertible in the sense that an agent holding $\{X\}K$ and the inverse of K is able to extract X. Other terms have other rules for invertibility. $\{X, Y\}$ is invertible to extract either X or Y without any additional information; $\mathtt{xor}(X, Y)$ is invertible to extract either X or Y given the other; $\mathtt{hash}(P)$ is invertible to extract P provided that the domain of hash is a type representing weak passwords (inversion is by guessing and comparing).

CAPSL provides a general, uniform way to express all of these properties and any introduced by users. They all can be expressed by use of axioms involving the relation *invertible(Term, Var, List)* for each invertible operator. The intended meaning of the invertible relation is that the first argument term can be inverted to extract the second argument variable if the variables in the list are held.

For example, invertibility of the public-key operator *ped* would be specified with an axiom:

```
IF keypair(K,SK) THEN invertible(ped(K,X),X,[SK]) ENDIF;
```

An operator may or may not have axioms that imply modes of invertibility. Conceivably the translator could infer some cases of invertibility from other axioms, but that seems difficult, and it is not unreasonable to ask the specifier to know and include invertibility information explicitly.

5 Translator Algorithm

Overview of translation

The translator from CAPSL to CIL has some commonplace tasks to perform, like parsing and type-checking, and it also performs the conceptually challenging task of unraveling a message-list protocol description into a set of rewrite rules. Type declarations are collected into a *symbol table*, which lists the declared types of variables and the type signatures of functions. Axioms from typespecs and environment specifications are consolidated into a single list.

Assumptions and goals are *localized* by associating them with states. Eventually, goals should be translated into proof objectives that are tied closely to the multiset model, for example, by converting them to testable invariants. A generalized attacker can also be incorporated into the multiset model, and it could be customized using information in the environment section.

The output of the CIL translation is therefore:

1. symbol table
2. axioms
3. localized assumptions
4. protocol rewrite rules
5. localized goals
6. attacker and environment information

Translator State

The CAPSL translator processes a message list to produce multiset rewrite rules. Only the translation of messages is described here. The complete translation algorithm deals with actions and conditional invocations, and can invoke previously declared equations when needed to compute the values of defined variables (from DENOTES declarations).

The translator is a finite-state machine. Its state is a set of role states, and its inputs are the messages in the message list, presented in order.

The state of a role is represented by a term $S(p, n, \mathbf{x})$ where p is a protocol variable of type Principal, the n is a state number, and $\mathbf{x}$ is a sequence of terms held by p. Most of the terms in $\mathbf{x}$ are protocol variables, but compound terms may be present as well.

For our purposes in describing the translation, we represent a message as a term $M(p, q, t\%r)$ where p and q are the variables representing the sender and receiver of the message, and $t\%r$ shows the sender's version t of the message content and the receiver's version r. In CAPSL, a message can have several fields, but for simplicity we assume here that t and r are single terms. If the message has more than one field, its content could be represented as a concatenation of these fields.

Computability and Receivability

We begin with some necessary terminology. In general, a boldface symbol is a sequence, so $\mathbf{x} = x_1, ..., x_n$ for some n. In some contexts we will also use $\mathbf{x}$ to refer to the set of its components. $R(p)$ denotes the symbol of type Role which corresponds to a symbol p of type Principal. A function $f(\mathbf{y})$ is *p-accessible* if f is not private (does not have the PRIVATE property) or f is private and $y_1 = p$. A variable t is *new* in the current translator state if t is a protocol variable, t is of type Nonce or has the FRESH property, and no other principal q has t in its current state. A set of new variables is also called new.

In defining computability of a term, we assume that some terms are held–this is the set G–and we derive the set of additional variables X that are needed to compute the term. The principal p is mentioned only because of the need to test accessibility.

Definition. t is *p-computable given G with X* if

1. $t \in G$ and $X = \emptyset$ or
2. t is a protocol variable and $t \notin G$ and $X = \{t\}$ or
3. $t = f(\mathbf{y})$, $f(\mathbf{y})$ is p-accessible, each y_i is p-computable given G with X_i, and $X = \bigcup_i X_i$.

We say that t is p-computable given G if t is p-computable given G with $\emptyset$. If Z is a set of terms, we say that Z is p-computable given G with $\bigcup_{t \in Z} A_t$ if each $t \in Z$ is p-computable given G with A_t.

Example. Consider $t := \mathtt{ped}(\mathtt{SK}(A), N)$. t is A-computable given $\{A\}$ with $\{N\}$ because `ped` is not private, and, although `SK` is private, $\mathtt{SK}(A)$ is A-accessible.

Definition. t is *p-invertible at i given G* if $t = f(\mathbf{y})$ and $\mathtt{invertible}(f(\mathbf{y}), y_i, Z)$ and Z is p-computable given G.

If a term t is a variable or constant (a function with no arguments), receiving it means to compare it with the terms in the held set G and add it to G if it is not there. If t is compound, it must be either computable or invertible, and in the latter case the components extracted from it are received recursively. This process enlarges G to H.

Definition. t is *p-receivable given G to H* if

1. t is p-computable given G and $H = G$, or
2. t is p-computable given G with $\{t\}$ and $H = G \cup \{t\}$, or
3. We have:

   (a) $t = f(\mathbf{y})$ is p-invertible at some j given G and

(b) $\mathbf{y}'$ is sequentially p-receivable given G to H', where $\mathbf{y}'$ is the maximum subsequence $y_{i_1}, ..., y_{i_k}$ such that t is p-invertible at i_j given G, and

(c) if t is p-computable given H' then $H = H'$ else $H = H' \cup \{t\}$.

Definition. $\mathbf{y} = y_1, ..., y_n$ is *sequentially p-receivable given G to H* if, for $j = 1, ..., n$, y_j is p-receivable given G_j to H_j, where $G_j = H_{j-1}$ and $H_0 = G$ and $H_n = H$.

The success or failure of the sequential receivability test depends on the order of the sequence of terms, since the held set G is augmented as part of the process. A more forgiving definition would be able to rearrange the order to find one that works, and it could be implemented by making several passes over the sequence.

Example. Consider $t := \mathtt{ped}(\mathtt{SK}(A), N)$. t is B-receivable given $\{A\}$ to $\{A, N, t\}$. Upon receiving t the agent in role B not only learns the nonce N but also the whole term t since t is not B-computable given $\{A, N\}$.

Rewrite Rules

A message $M(p, q, t\%r)$ gives rise to two protocol rewrite rules, one for p to send the message t, and one for q to receive r. Each protocol rewrite rule is generated by a translator state transition. A transition associated with sending the message affects only the sender-role state, and the one associated with receiving the message affects only the receiver-role state.

A schema is a way of presenting a set of translator transitions in a parameterized form, independent of the particular state number and term sequence. There is a Send schema for the sender-role transition and a Receive schema for the receiver-role transition. A schema may specify conditions on the state transition; if they are not satisfied, the transition fails, and so does the translation. A schema ends with a protocol rewrite rule. Protocol rules have the form specified in the CIL abstract syntax, as `rule`([*left-facts*], [*new-vars*], [*right-facts*]).

The Send schema says that if the message is computable, possibly with a set of new variables, the sender can transmit the message. The sender must also hold the identity of the receiver. The Receive schema says that if the message content is receivable with learned terms A, the receiver accepts the message and adds the terms in A to its state. Because of space limitations we only present the receive schema here.

Notation. If A is a set of variables, $\mathbf{A}$ consists of the elements of A written as a sequence, in some arbitrary but consistently chosen order.

Receive schema:
Current state: $S(q, n, \mathbf{x})$
Message: $M(p, q, t\%r)$
Condition: r is q-receivable given $\mathbf{x}$ to H and $A = H - \mathbf{x}$
Next state: $S(q, n+1, \mathbf{xA})$
Rule: $\mathtt{rule}([\mathtt{state}(R(q), n, \mathbf{x}), \mathtt{msg}(z, q, r)], [], [\mathtt{state}(R(q), n+1, \mathbf{xA})])$

The receiver of a message cannot see the sender's address. Thus, we assume an arbitrary sender variable z of type Principal.

Example. Given the translator state `S(B,2,B,A)` and the message `M(A,B,ped(SK(A),N))`, the new translator state is `S(B,3,B,A,N,ped(SK(A),N))` and the following CIL rule is generated:

```
rule([state('B',2,B,A),msg(Z,B,ped(SK(A),N))],
     [],
     [state('B',3,B,A,N,ped(SK(A),N))]).
```

6 Maude Implementation of the Translator

Maude [CELM96] is a multi-paradigm executable specification language based on rewriting logic [Mes92]. The Maude system, its documentation, and a collection of examples is available on the web at `http://maude.csl.sri.com`. Rewriting logic has proved to be a general and flexible semantic framework for specifying a wide variety of models of concurrency [Mes92], languages and logics. The adequacy of rewriting logic as a logical framework in which other logics can be represented has been illustrated by means of various examples, including equational, Horn, and linear logic (see [MOM93]). Exploiting the expressive capabilities of rewriting logic allow us to automate the passage from CAPSL specifications to CIL specifications. For this purpose we define the grammar of both CAPSL and CIL in Maude, and we formally specify the translator in Maude via a set of equations, making explicit each translation step.

A CAPSL abstract datatype, i.e., the CAPSL grammar equationally specified in a Maude module, is defined with the help of several sorts, subsort relations and operators.

```
mod TRANSLATOR is
 sorts CAPSLSpec ComponentList Component Typespec Protocol
       Environment Ident ...
 subsort Typespec Protocol Environment < Component < ComponentList .
 op specification : ComponentList -> CAPSLSpec .
 op typespec : Ident Decls Axioms -> Typespec .
 op protocol : Ident Decls Asserts PhraseList Goals -> Protocol .
 ...
```

A CAPSL specification is a list of components. Protocols, typespecs and environment specifications are components. Their definitions are very close to the abstract syntax of CAPSL. Sorts and operators for CIL specifications are set up in a similar way.

We illustrate the implementation of the translator algorithm with the help of p-computability of a term and the production of a CIL rule for the sender view of a message. For a given term t, an agent identifier p, and the current state G, the implementation returns a boolean value, and, in case t is p-computable, a set of new variables X. We use the current rule list as the basis to derive the set of terms G held by an agent. The implicitly assumed symbol table (a list of declarations) is an explicit parameter in the Maude implementation.

```
op _is_computableGiven_SymbolTable_ : Term Ident RuleList DeclList
                                      -> [Bool,TermList] .
eq t is p computableGiven G SymbolTable ST
  = if t isIn G then [true, nil]
    else if t isProtocolVariableNotIn G SymbolTable ST then [true, t]
         else if t is p AccessibleFunctionSymbolTable ST
              then [and(prj1(t is
                             p computableForAllArgumentsGiven G
                             SymbolTable ST),
                    union(prj2(t is
                               p computableForAllArgumentsGiven G
                               SymbolTable ST))]
              else [false, nil] fi fi fi .
```

Proper definitions for boolean functions such as `_isIn_ : Term RuleList -> Bool` are given in the implementation. The function `_is_computableForAllArguments_Given_SymbolTable_` delivers for each argument a Boolean and list of learned terms. The result of the computability function is the conjunction of boolean values and the union of all learned terms (`prj1, prj2` are projection functions).

The generation of the receiver rule from a message is defined in Maude with the following conditional rule:

```
op receiverRuleForMessage_CurrentState_SymbolTable_ : Message RuleList DeclList
                                                       -> RuleList .
```

```
ceq receiverRuleForMessage message(p,q,t) CurrentState G SymbolTable ST
   = rule([lastStateOf q In G, msg(z,q,t)],
          [],
          [update (lastStateOf q In G)
           With prj2(t is q receivableGiven G SymbolTable ST)])
   if prj1(t is q receivableGiven G SymbolTable ST) .
```

If t is q-receivable given G to H, then `_is_receivableGiven_SymbolTable_` returns the tuple [true,H].

7 Issues

CAPSL, CIL and the translation between them are designed to address important issues in cryptographic protocol specification for analysis purposes. Type specifications in CAPSL and their use for introducing new operators and principal subtypes address the need for generality. The linear-logic/term-rewriting style for CIL meets the need for a clear analysis-level modeling semantics and a pattern-matching transition rule style. Abstract specification of the translation, with an executable version in Maude, meets the need for unambiguous CAPSL semantics.

There are areas still under development, especially surrounding the specification and semantics of goals. Due to the experimental activity, design choices are constantly being reexamined and debated. Two are discussed further here: invertibility and session identifiers.

Invertibility

Generality is an important goal for CAPSL. The "invertible" relation brings us closer to this goal by making it possible to axiomatize the invertibility properties of new operators. We think it is a reasonable task for specifiers to include this property for operators they define, but we also have considered whether invertibility properties can be inferred from the functional axioms.

For example, the axiom `sd(K, se(K, X)) = X` in the symmetric key typespec implies an invertibility relation for `se`. In the future, we might be able to recognize opportunities like this so that the analyst will not need to specify invertibility explicitly.

On the other hand, invertibility is not always apparent. Hash functions on guessable passwords are effectively invertible, but there is no axiom for hash functions from which to infer that fact. Even when suitable axioms are present, invertibility is not always easy to determine. Invertiblity of *xor* for both arguments can be deduced from the combination of its commutativity, associativity, and cancellation axioms, but is there a decision procedure for this?

Session Identifiers

An initialization rule like $\longrightarrow$ `state('A',`$0, A, B$`)` can create any number of agents playing role 'A' in the protocol, but they will not be distinguishable from others using the same principals A and B. Protocol implementations distinguish different agents playing the same role for the same principal through some session identifier such as a port number or sequence number.

From a confidentiality analysis point of view, there is no loss of generality in omitting a session identifier in the protocol model, since the attacker model includes the ability to forge false session identifiers.

From an authentication point of view, however, the absence of a session identifier means that we might not have a way to express security goals asserting the agreement of two communicating agents on a supposedly shared data item such as a session key. Some sort of nonce is necessary to ensure that Alice in this session doesn't share a value held only by Bob in some other, perhaps old, session. A session ID is needed to state an agreement property for a protocol that does not have any explicit nonce. The urgency of this need, however, is cast into doubt when one considers that a protocol without an explicit nonce could not satisfy an agreement property, since it would have no way of enforcing or checking it.

In any case, it is easy to include a session ID in the translation if desired. Just add one to every state and message; it is existentially generated in the initialization rule.

Plans

To make CAPSL useful to protocol analysts, we need to make translation tools as well as documentation available. Our plans call for a translator from CAPSL to CIL implemented in more than one transportable form. The capabilities of the translator for representing goals and an attacker model will be gradually expanded. We should soon have examples of CIL-to-tool-translation, both our own and others.

References

[BMM99] S. Brackin, C. Meadows, and J. Millen. CAPSL Interface for the NRL Protocol Analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology, ASSET '99, Dallas* , 1999. To appear.

[Bra97] S. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Symposium on Network and Distributed System Security*. Internet Society, February 1997.

[Car94] U. Carlsen. Generating formal cryptographic protocol specifications. In *IEEE Symposium on Research in Security and Privacy*, pages 137–146. IEEE Computer Society, 1994.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Rewriting Logic and Its Applications, First International Workshop, Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996*, pages 65–89. Elsevier Science B.V., Electronic Notes in Theoretical Computer Science, Volume 4, `http://www.elsevier.nl/locate/entcs/volume4.html`, 1996.

[Cer99] Cervesato, I. and Durgin, N.A. and Lincoln, P.D. and Mitchell, J.C. and Scedrov, A. A metanotation for protocol analysis. In *11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1999. To appear.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998. `http://www.cs.bell-labs.com/who/nch/fmsp/index.html`.

[Low98] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil97] J. Millen. CAPSL: Common Authentication Protocol Specification Language. Technical Report MP 97B48, The MITRE Corporation, 1997.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. CSL Technical Report 93-05, SRI International, Computer Science Laboratory, Menlo Park, CA, August 1993.